Improving File System Usability, Performance and Reliability with Magnetic RAM

Ethan L. Miller
Storage Systems Research Center
Institute for Scalable Scientific Data Management
University of California, Santa Cruz

What is Magnetic RAM?

- Magnetic RAM (MRAM) is
  - Random access: speed comparable to DRAM
  - Non-volatile
- MRAM is byte-accessible
  - No need to read or write full blocks
- MRAM doesn't suffer wear
  - No need for wear-leveling
  - No limit on number of writes
- MRAM is now in production
  - Freescale has been making chips for over two months
- MRAM is (currently) expensive
  - Eventually cost comparable to DRAM
  - New product -> expensive!

Why file systems for MRAM?

- File systems traditionally use disk as non-volatile storage
  - Large blocks / transfer sizes
  - Expensive seeks
  - Metadata size relatively unimportant
- MRAM provides
  - Long-term storage
  - Byte-addressability with low seek time
  - High-speed access to complex metadata
- Problems:
  - Cost is a major issue (≥ DRAM)
  - File systems aren't designed to take advantage of MRAM
- What should a file system for MRAM look like?

ISSDM

What should we keep in MRAM?

- Memory (non-volatile or otherwise) is expensive
  - Keep small items in it?
  - Keep recently used items in it?
- Large (data) transfers don't benefit as much from memory residence
  - Most transfers are large and sequential
  - Latency can be hidden with prefetching and writebehind
- Metadata is perfect for MRAM!
  - Small transfers (often a word or two)
  - Still very large: about 1% of total file system size
  - Reduce memory demands with new metadata structures and compression
  - Allows the construction of richer metadata structures that might require non-sequential accesses

Outline

- Introduction to MRAM for file systems

- LiFS: Linking File System
  - More effective searching and organization
  - Made possible by MRAM's speed

- Compressing metadata in MRAM
  - Making the most of a scarce resource

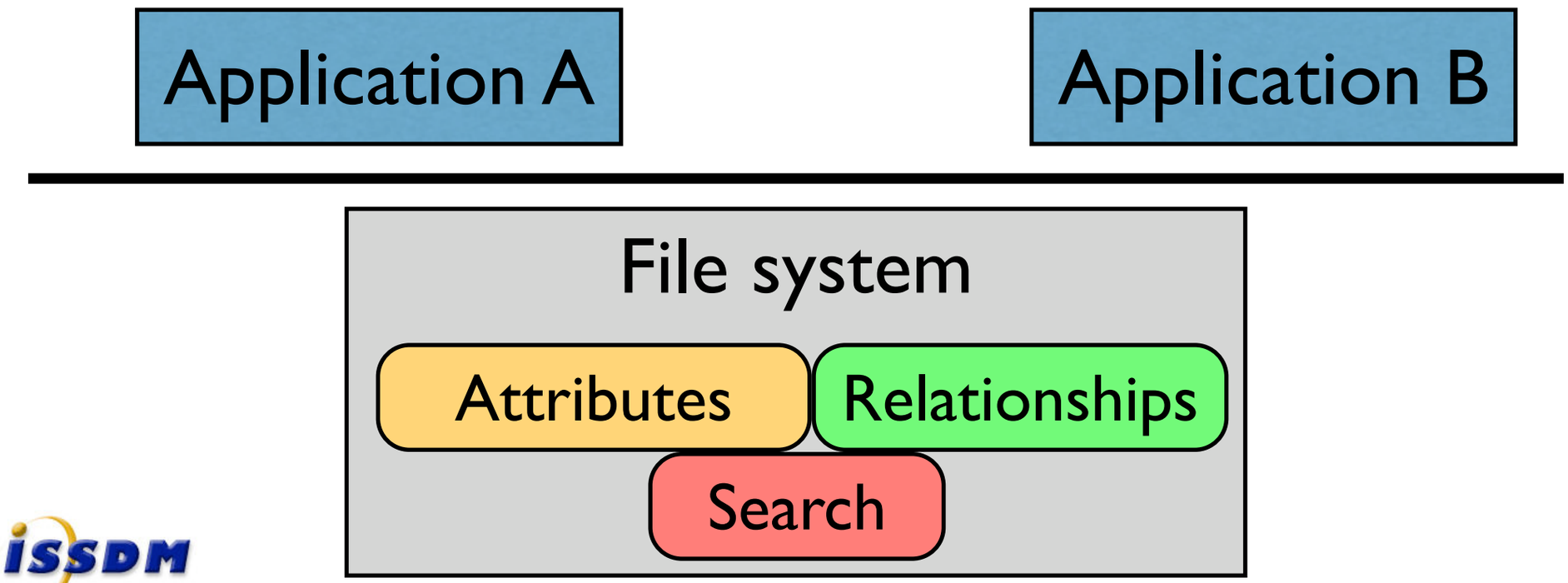
- MRAM reliability techniques
  - Making MRAM safe for file systems

The problem with metadata

- The number of files is dramatically increasing
  - Disk capacity is far larger
  - Applications like to use lots of files
- How can we organize them?
  - Directories?
    - Good model for few files, but not for billions!
    - Difficult for general directed graphs: files are typically in only one directory
  - Applications?
    - Many apps manage their own files
    - Works (somewhat) but makes sharing difficult

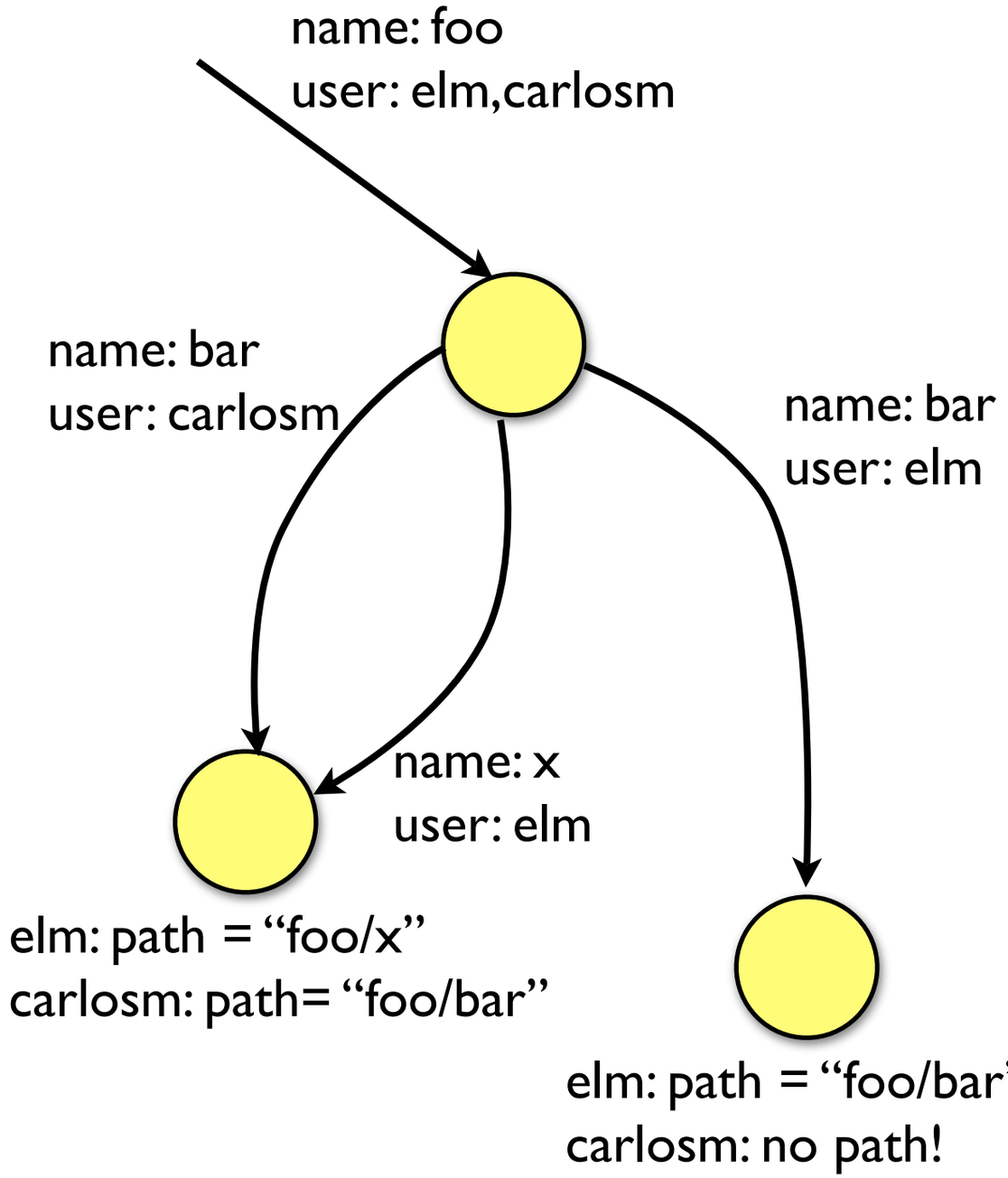
The solution: attributes & links

- Extend application approach into the file system
  - Provide primitives to manage the relationships
  - Allow multiple apps to use the same files and links
- Sharing is easy
  - File system maintains the relationships
- Searching is now possible…



MRAM makes this possible

- Directed links between files to show relationships
- Links have attributes
  - Express the type of relationship between the files
  - Describe the link itself
- Lots of links means lots of "seeks"
  - MRAM makes this fast
- Links are small
  - MRAM has low latency
- Searching can be slow
  - Keep the indices in MRAM



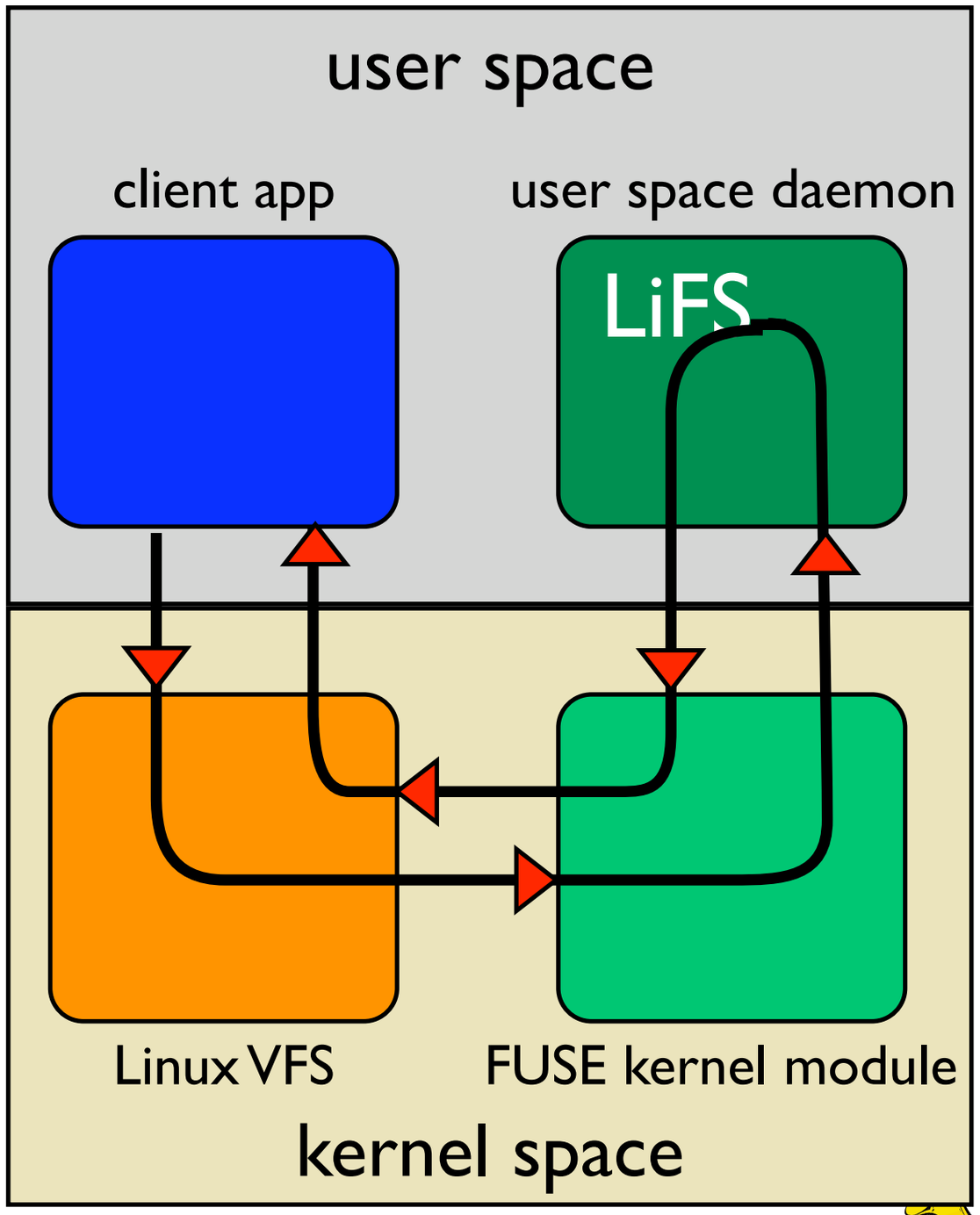
New system calls

System Call	Function
`rellink`	Create relational link
`rmlink`	Remove relational link
`setlinkattr`	Set attributes on link
`openlinkset`	Return handle to all the links from a file
`readlinkset`	Get name and attributes of next link in the set

ISSDM

Implementation

- FUSE: maps VFS calls back into user space
- MRAM: system memory locked into DRAM
  - Not yet using MRAM...
- Custom MRAM allocator with fixed-size pools
  - Efficient to allocate and free small objects
- Optimizations
  - String table
  - Full path name cache



Evaluation

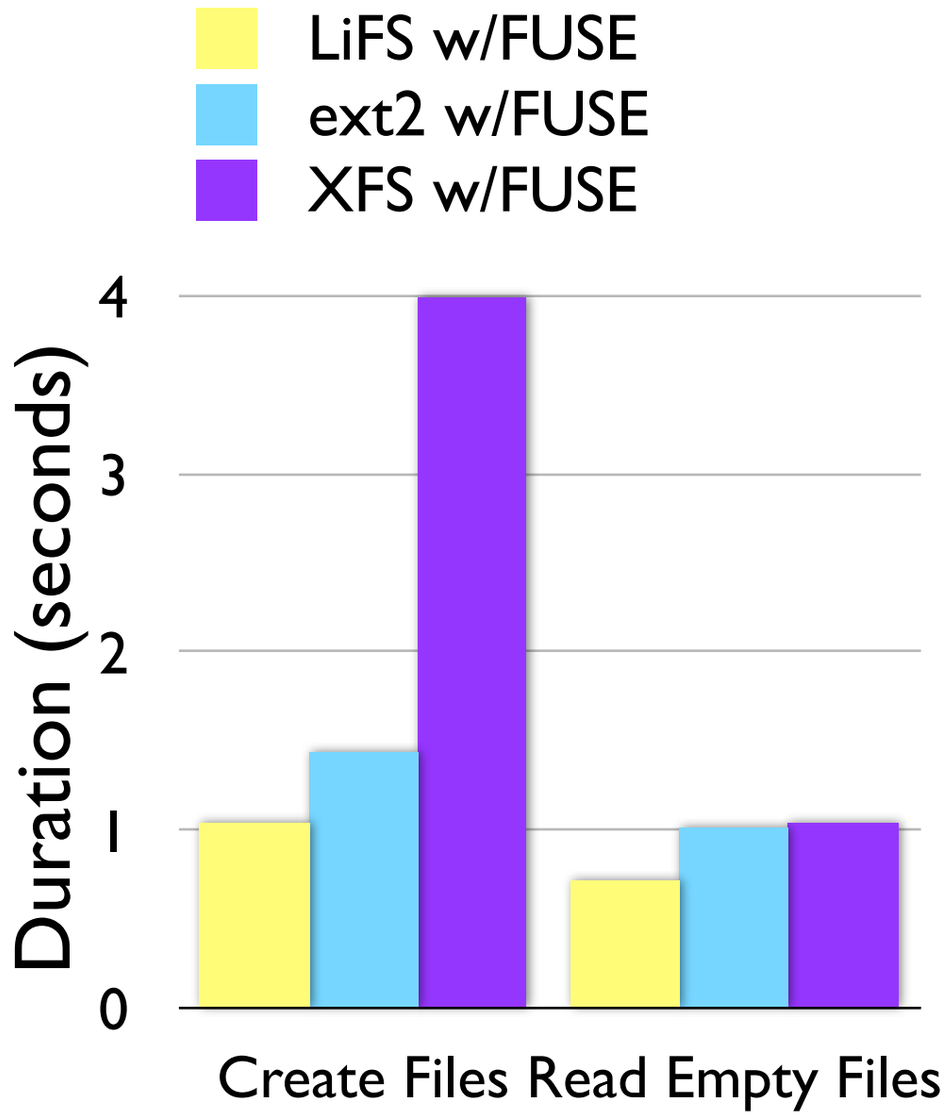
- Metrics
  - Traditional FS operations: compare to other file systems
  - New FS operations: scalability
  - FUSE overhead
- Experimental setup
  - Sun workstation running Linux 2.6.9-ac11
  - AMD Opteron 150 @ 2.4 GHz
  - 1 GB DRAM

ISSDM

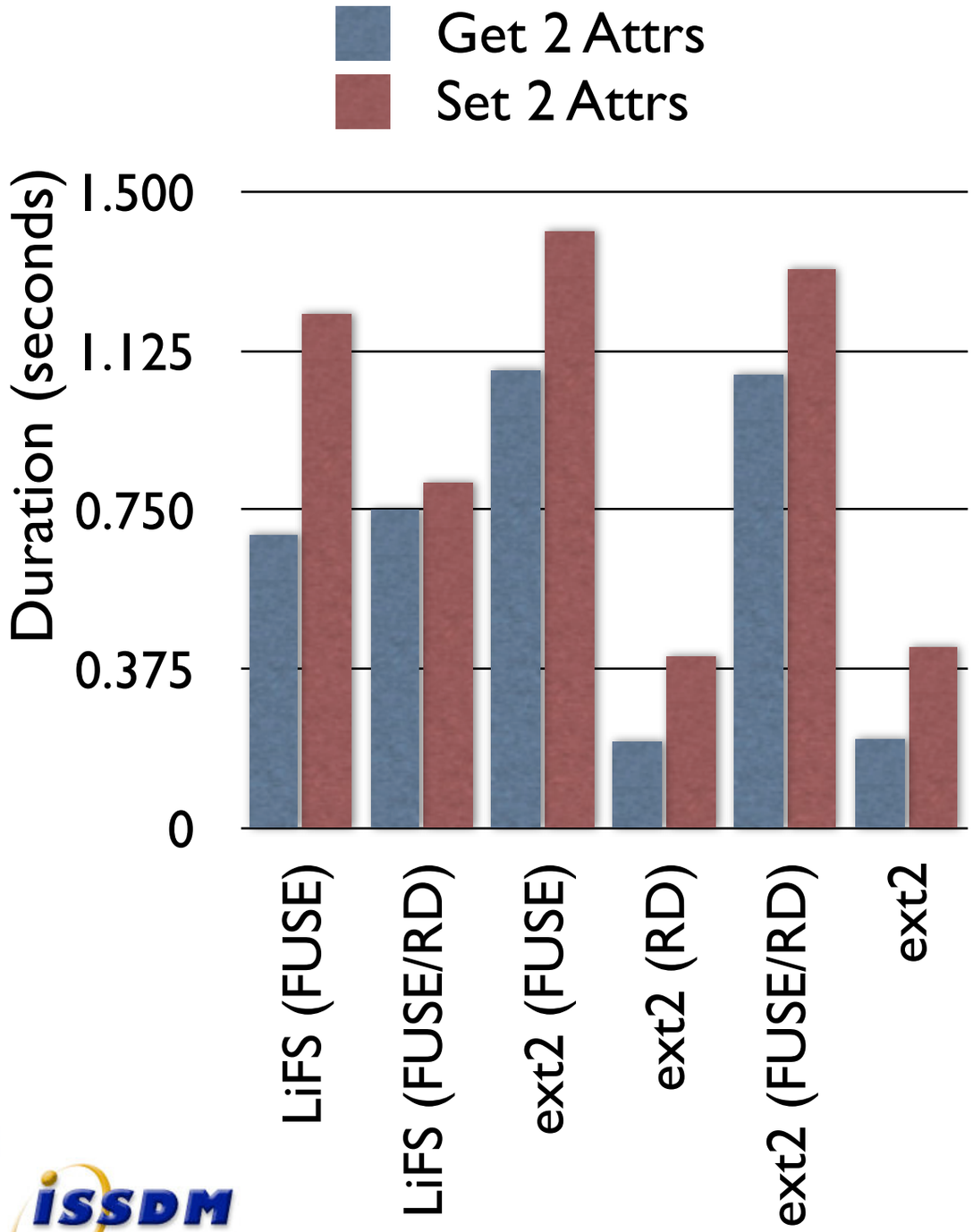
Performance: Files

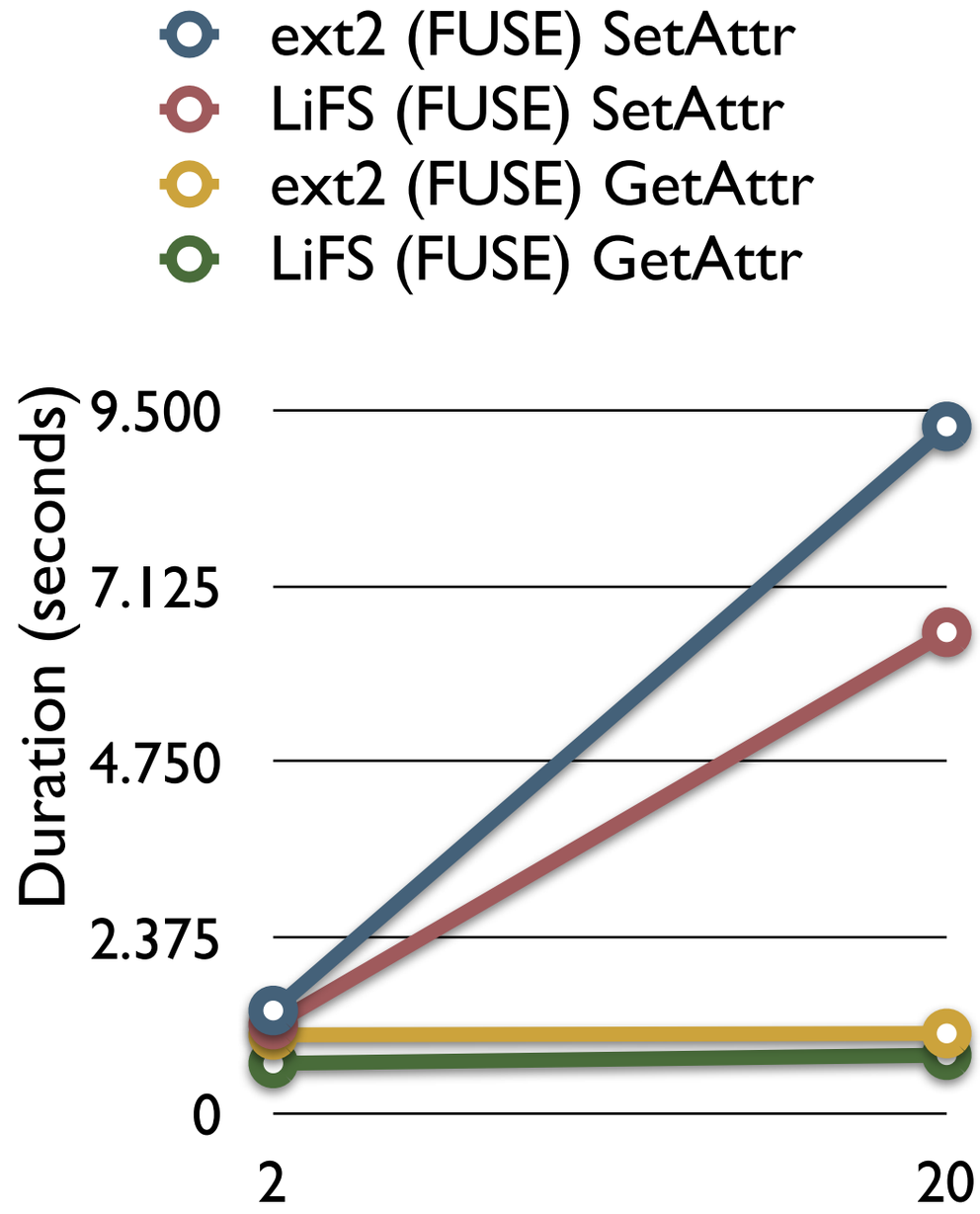
- Create a directory tree with empty files
  - 15620 files
- Read all of the files
- File systems are "fresh"

- LiFS is competitive

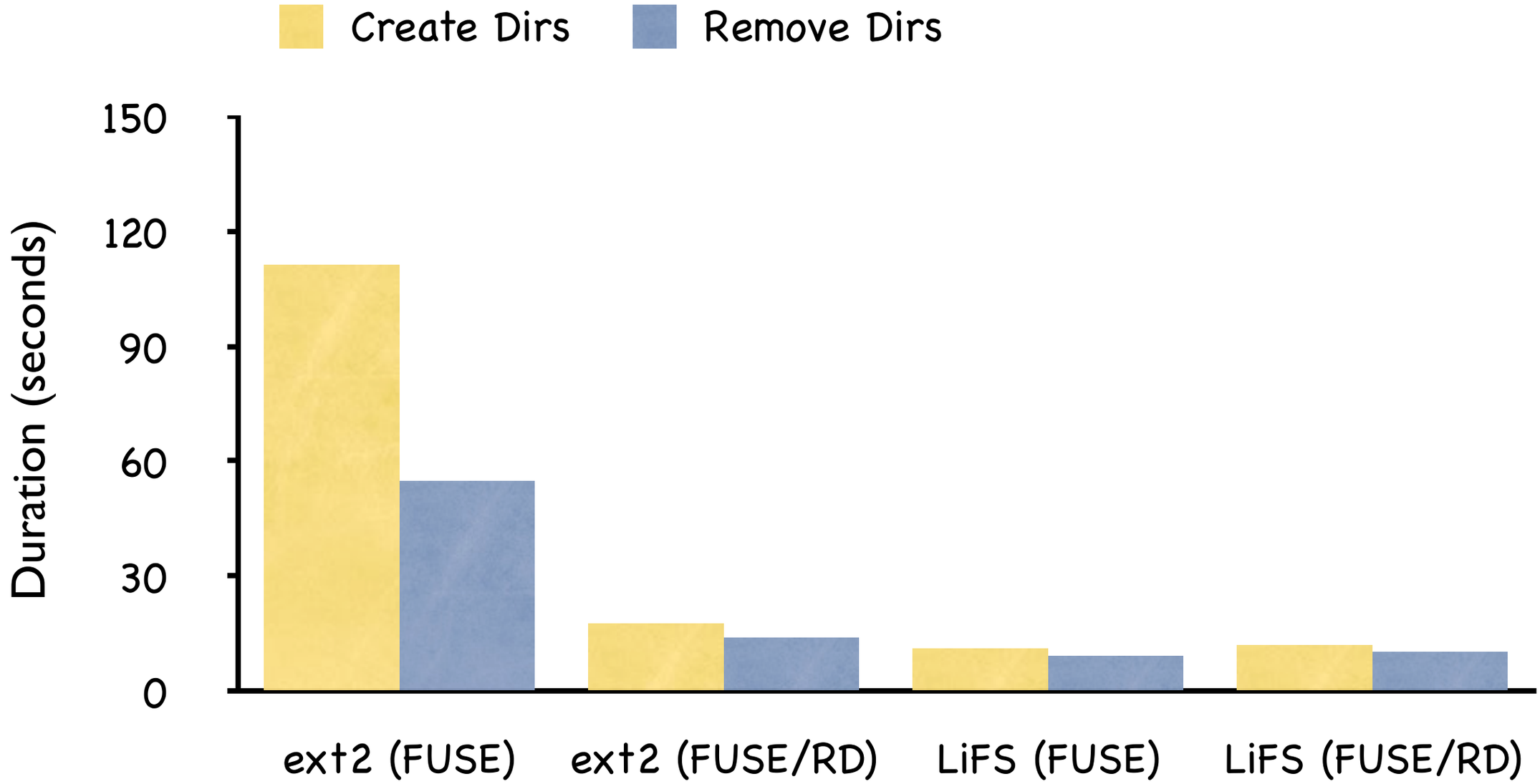


Performance: File Attributes





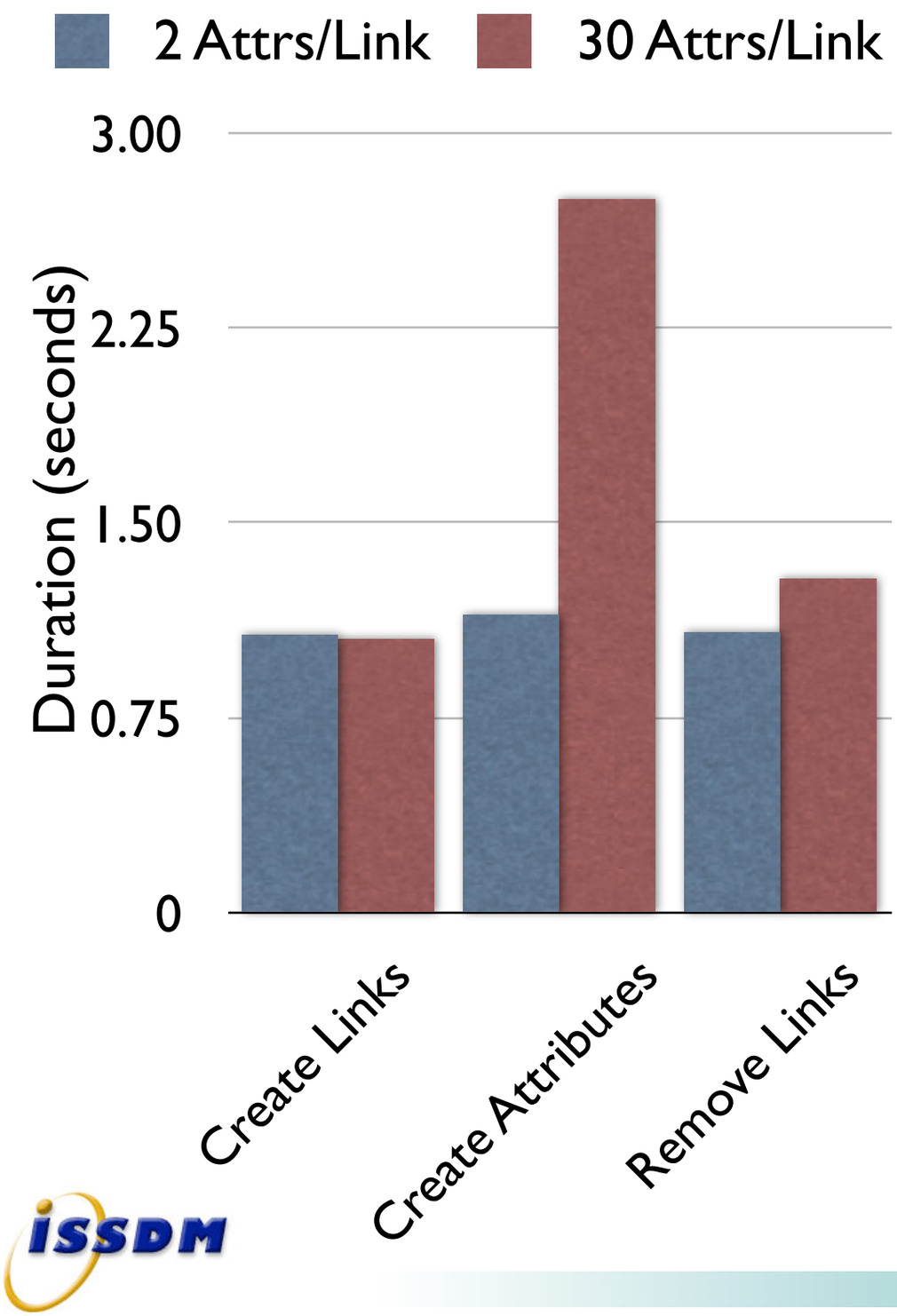
Create / Remove Directories



111,110 directories

LiFS outperforms ext2 with FUSE and RAM disk

Create / Delete LiFS Links



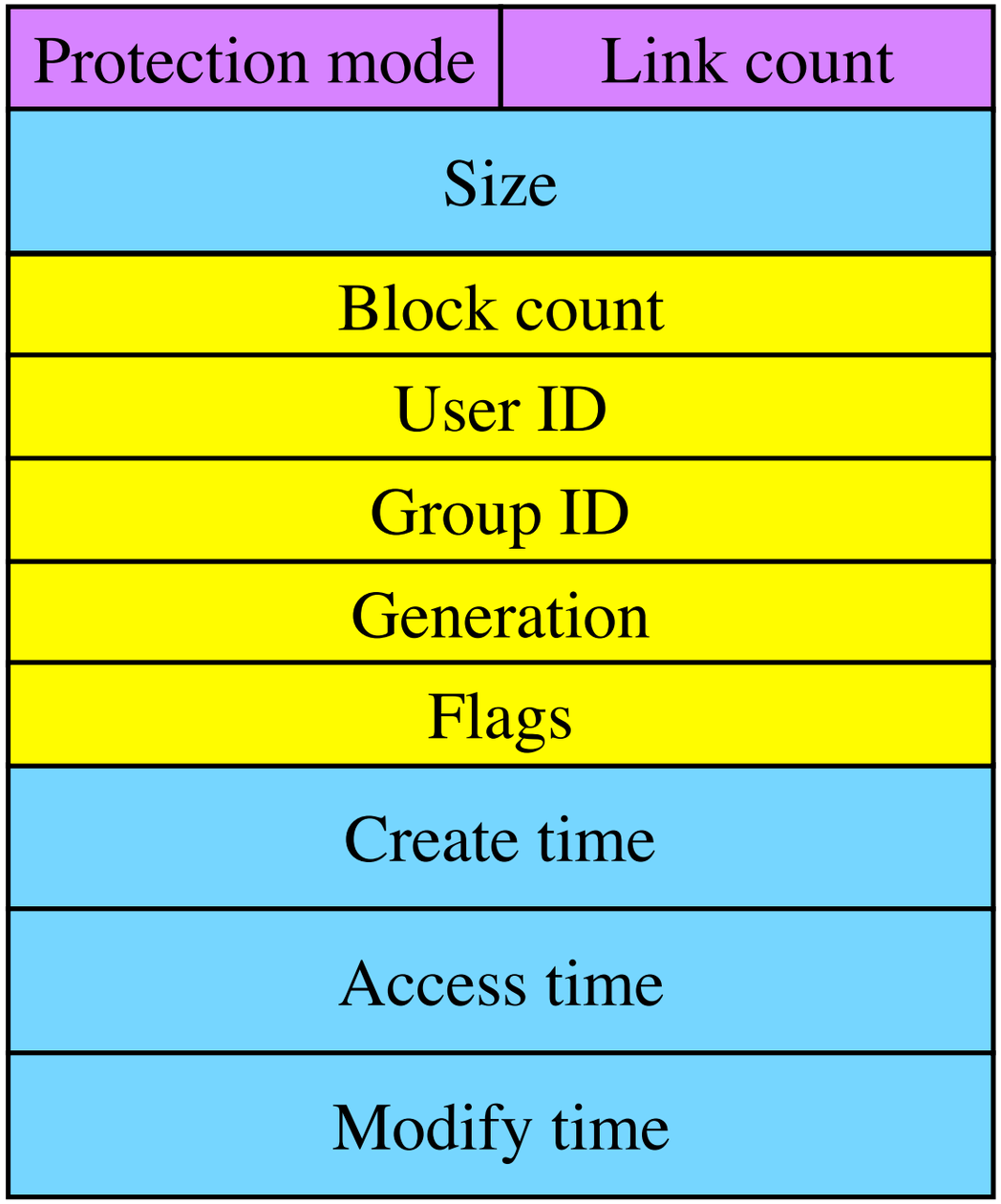
- Test on 15,620 files
- Processed 15,620 random links
- More attributes make link identification slower
  - Need to traverse structures to identify desired link

Compressing Metadata in MRAM

- LiFS promises lots of additional functionality, but…
- MRAM is expensive!
  - Currently, much more than DRAM
  - Eventually, costs drop to about DRAM costs
- Important to save space in MRAM if possible: compress metadata
  - Reduces MRAM requirements
  - May improve speed by reducing the amount of data moved
  - Byte-accessible MRAM makes this feasible

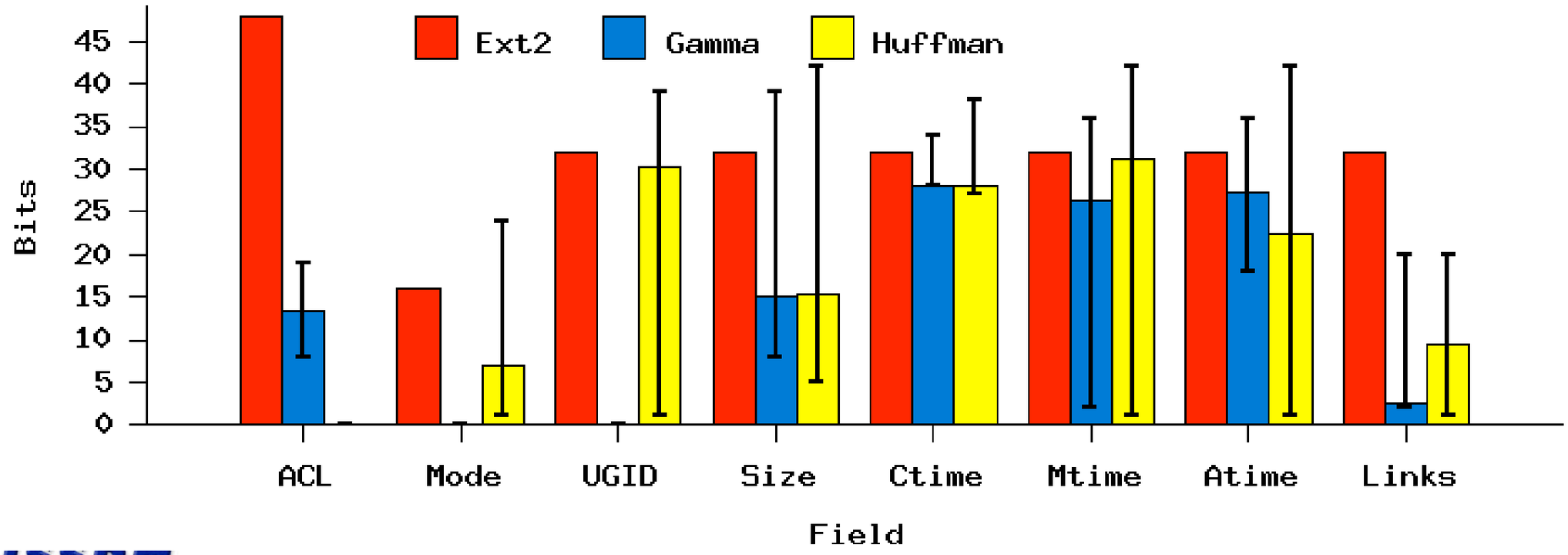
Metadata in Unix

- Metadata is stored in inodes
  - Timestamps
  - Ownership
  - File size
  - Link count
- Directories point to inodes
  - Inodes themselves don't contain names
- Total size is about 54 bytes
  - Times, size are 64 bit fields



Compressing metadata

- Most metadata is compressible
  - Integers have small values (link count, size)
  - Times can be expressed as offsets
  - Permissions can be table-based
- Compression is effective
  - 15–20 bytes per inode on typical file systems (factor of 2.5–3.5)
  - Inodes are variable-sized: compression rate varies
- Can this be used in a real file system?
- How does it affect performance?



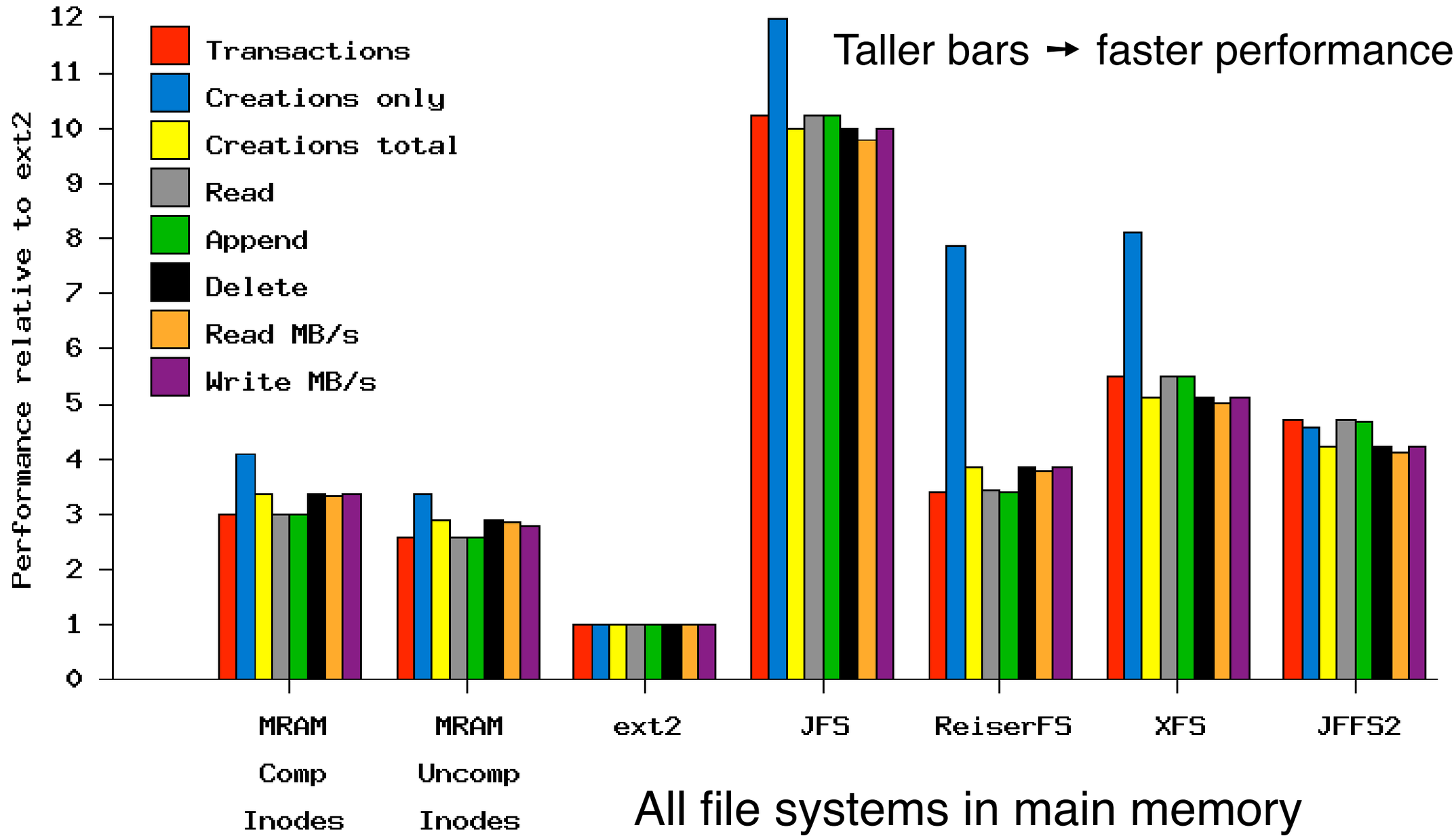
Inodes and file data

- Fields in the inode are gamma compressed
  - Small numbers compress very well
  - Timestamps are encoded as deltas from earliest time
- Permissions are table-based
  - Table contains all unique combinations of <*user ID*, *group ID*, *permissions*>
    - Most common combinations stored early in the table
  - Location in table of this inode's permissions is gamma-compressed
  - Table could easily be extended to handle ACLs…
- File data stored in compressed blocks
  - Linked list of block pointers
  - Random seeks not slow: pointer chasing is in RAM
  - Hybrid on-disk / MRAM-based system will need some minor changes

Separation of MRAM and DRAM

- Most memory-based file systems simply use the caching structures as "permanent" storage
  - Very fast: almost no copying
  - Inefficient: caches designed for very fast lookup at the cost of higher memory usage
  - Unsuitable for MRAM / DRAM systems
    - Must build the whole system from MRAM
    - No way to "start clean"
- MRAMFS explicitly copies from non-volatile memory to DRAM
  - Metadata and data copied back and forth
  - Can be slower for large operations
  - May be more reliable: fewer code points where permanent store is accessed and modified

Postmark benchmark results



All file systems in main memory
All file systems faster than disk-based

Postmark results

- All of the file systems run in-memory
  - Ext2 slowest by far: sequential directory handling
  - JFS very fast
  - MRAMFS comparable to ReiserFS
  - MRAMFS slightly slower than XFS, JFFS2
- All but JFFS2 are uncompressed
  - Memory usage higher by a factor of about 3
  - JFFS2 compresses blocks: not as efficient for inodes
- Compressed inodes faster in MRAMFS!
  - Less metadata copied: faster performance
  - Compression can provide speed increase by avoiding accesses to (relatively) slow DRAM

Compression reduces MRAM demand

- File systems will run much faster with non-volatile memory
  - Don't need to read metadata from disk
  - Don't write metadata back to disk
  - Fast access to parts of an inode
- Compression of metadata can reduce memory usage and thus cost
  - Factor of 2–3 or more
  - May save time, too: less data movement between (relatively) slow main memory and cache
- Alternate file structures can make efficient use of MRAM and other non-volatile memories
  - Relatively small allocation chunks
  - Simple, reliable file system design
  - No need for complex structures that may introduce bugs

Making MRAM file systems safe

- MRAM advantages
  - Fast metadata access
  - Richer metadata structures
  - Reduces sync and transfer overhead
  - File system log for fast recovery
  - Fast, on-line consistency checking
- MRAM disadvantages
  - Wild writes from buggy software
    - Potentially much less "control" than with disk-based file systems: no centralized I/O access routines
  - Bit flips on MRAM
- How can this corruption can be prevented?

Solution: Two-Level Approach

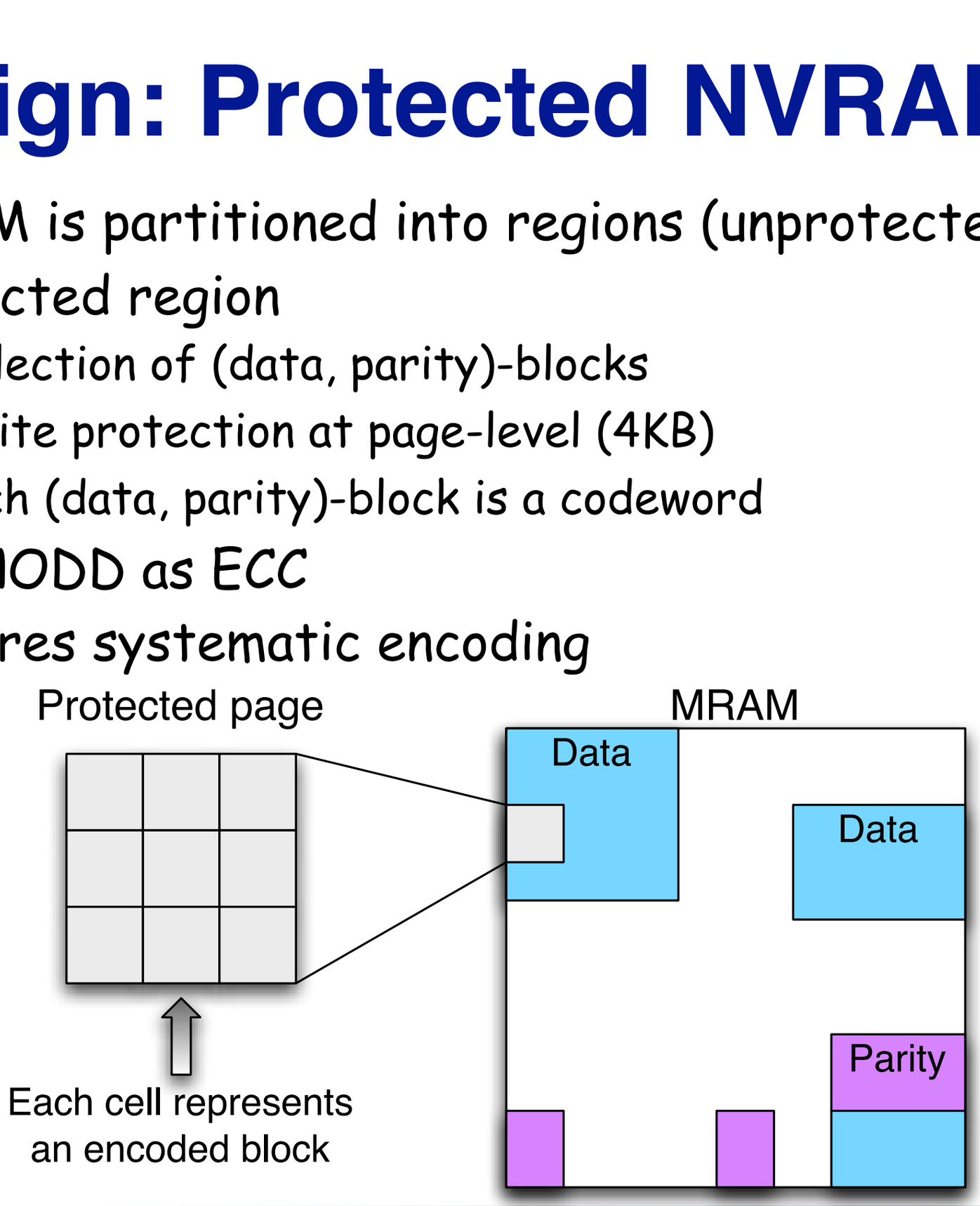
- Attack the problem on multiple levels
  - Guard against memory corruption due to OS mistakes
  - Check file system operations to make sure that they're correct

- Memory consistency
  - Page-level write protection
  - Error Correcting Codes (carefully chosen)
- File system consistency
  - Log/periodically check structure changes
  - On-line checks (log replay)

ISSDM

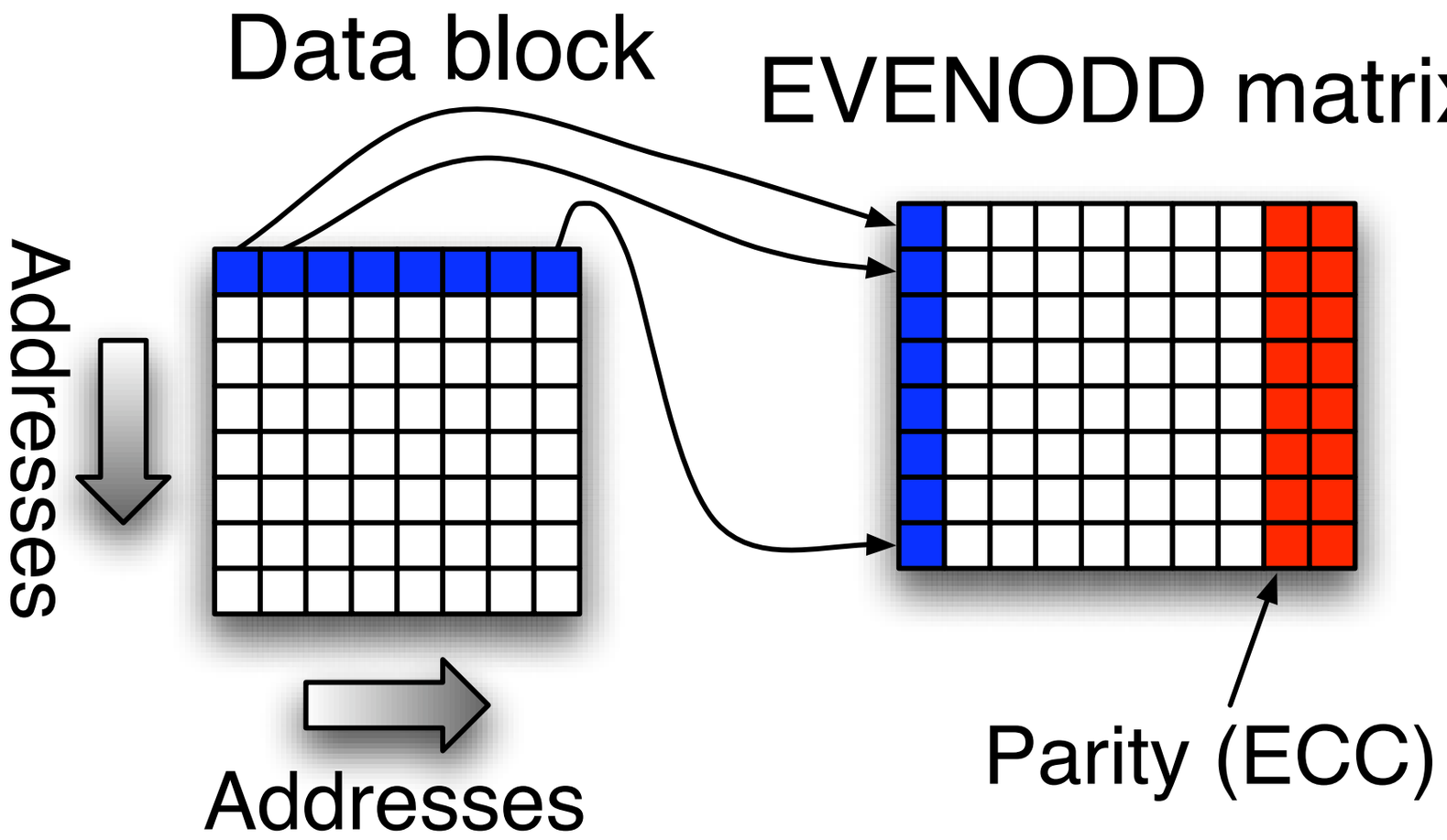
Design: Protected NVRAM

- MRAM is partitioned into regions (unprotected/protected)
- Protected region
  - Collection of (data, parity)-blocks
  - Write protection at page-level (4KB)
  - Each (data, parity)-block is a codeword
- EVENODD as ECC
- Requires systematic encoding



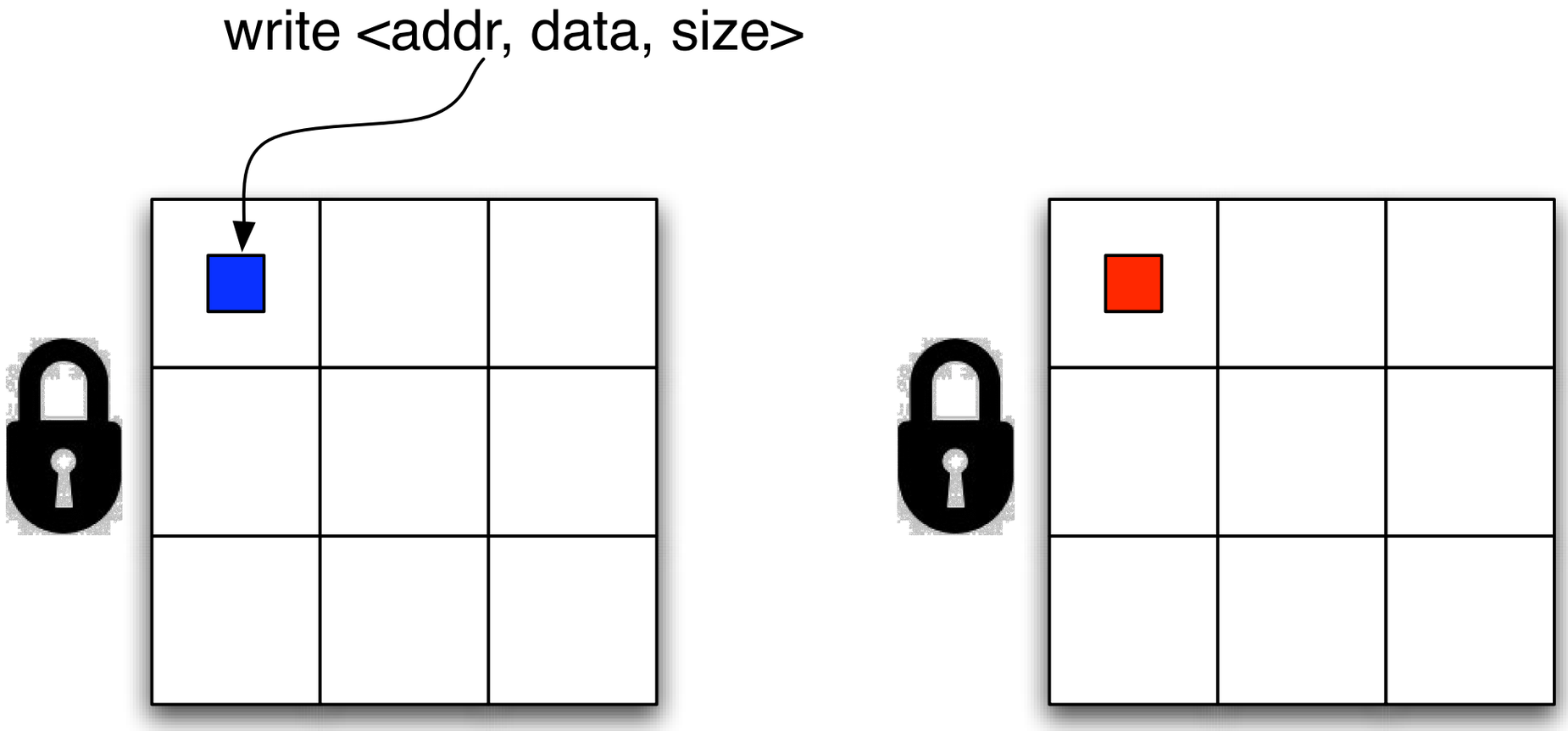
Design: Mapping Data to EVENODD

- Visualize the data block as a 1D array mapped to 2D array
- Can tolerate any one column error (bursts)



Write Algorithm: identify location

Step 1: Identify address and write size

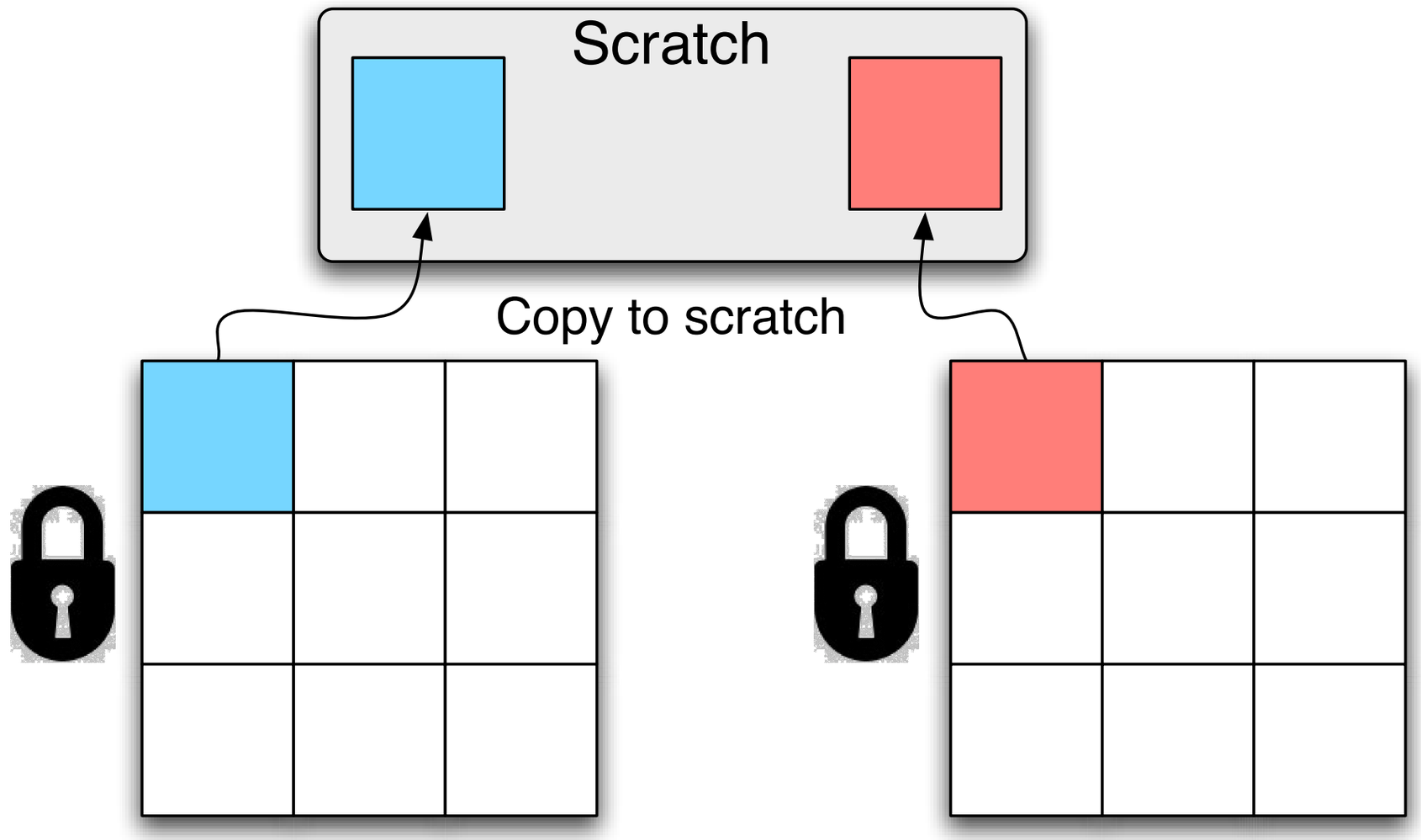


Data Page

Parity Page

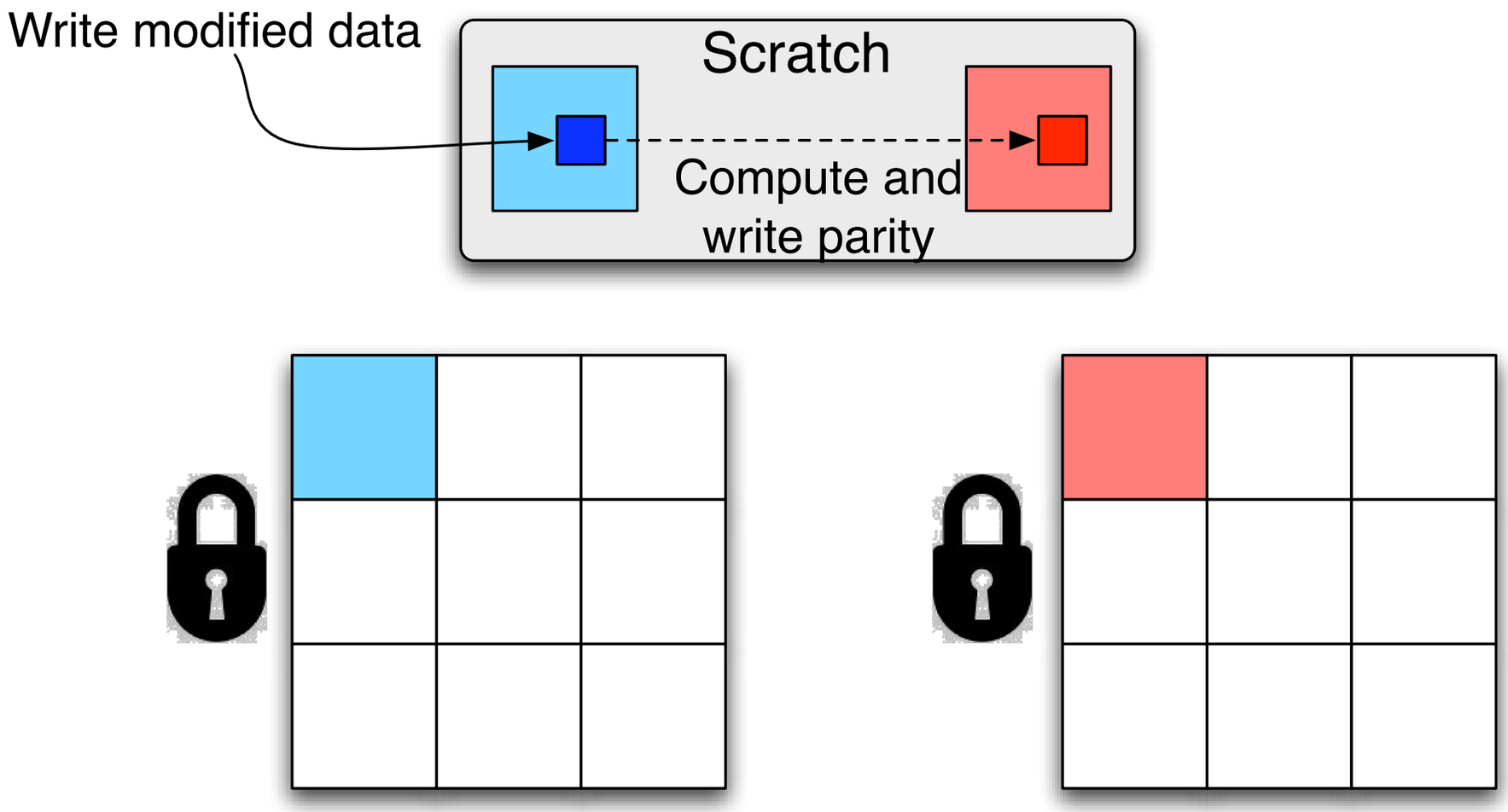
Write Algorithm: copy to scratch

Step 2: Copy corresponding data/parity sections out to "scratch"



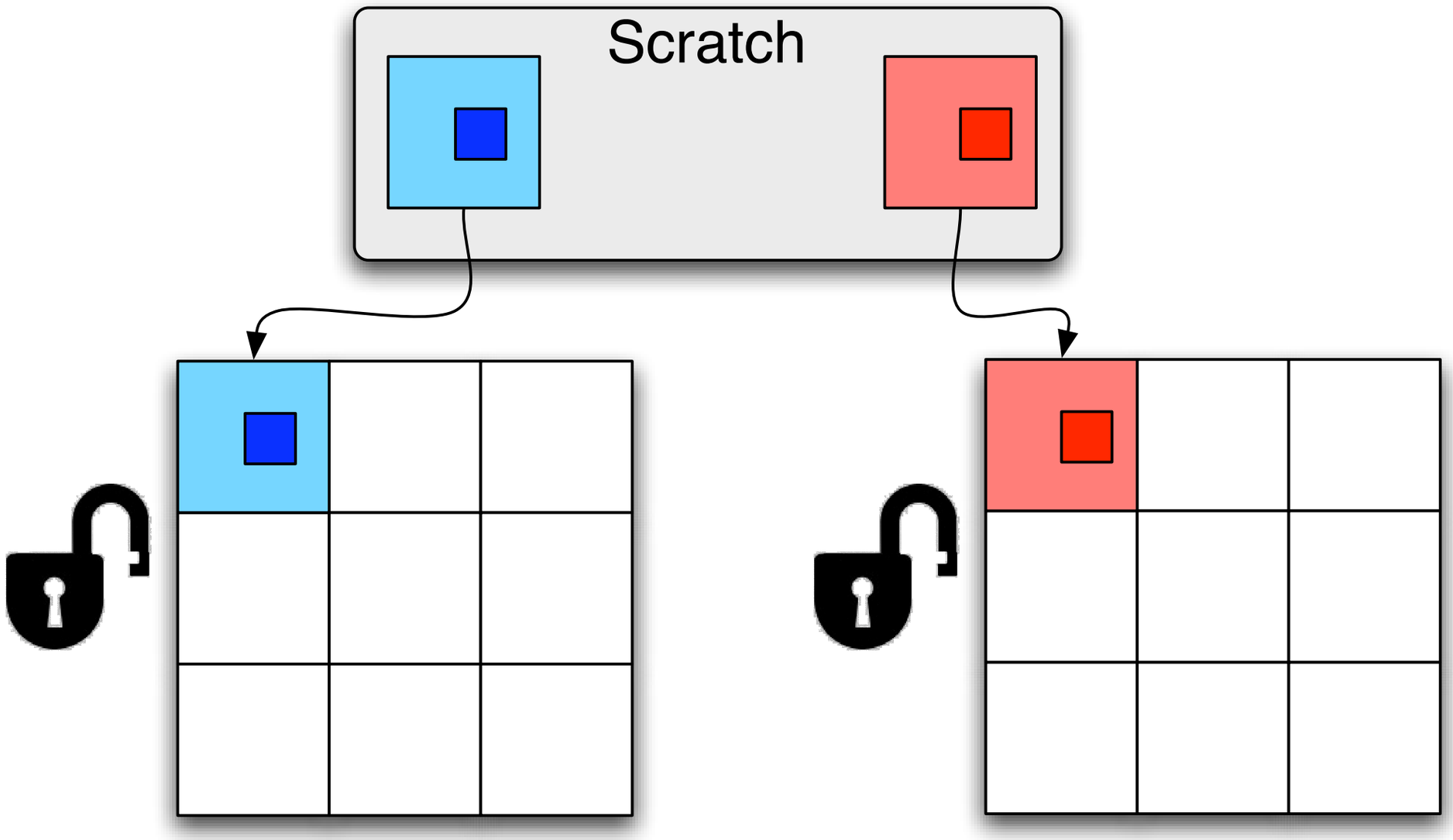
Write Algorithm: update in scratch area

Step 3: Write change in scratch region, compute parity



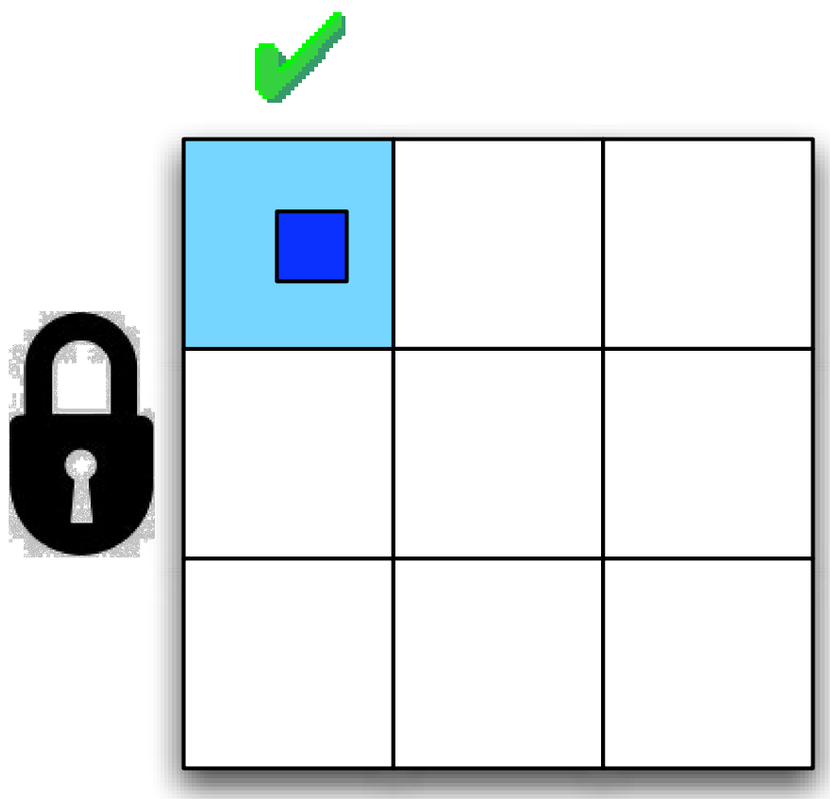
Write Algorithm: write back changes

Step 4: Unlock pages and write changes (vulnerable)

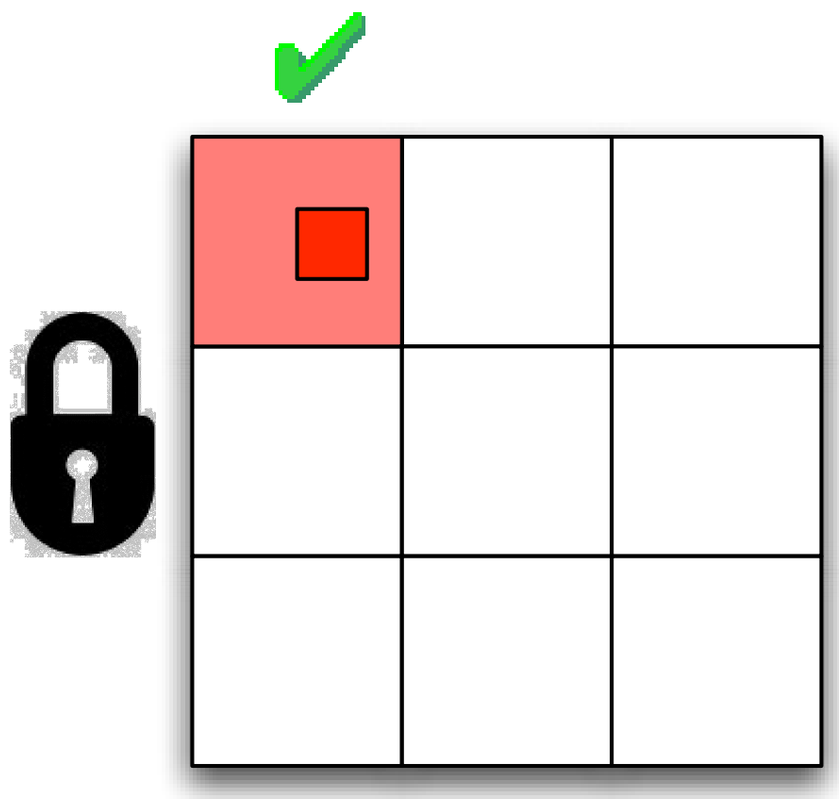


Write Algorithm: check writes

Step 5: Lock pages and check for errors



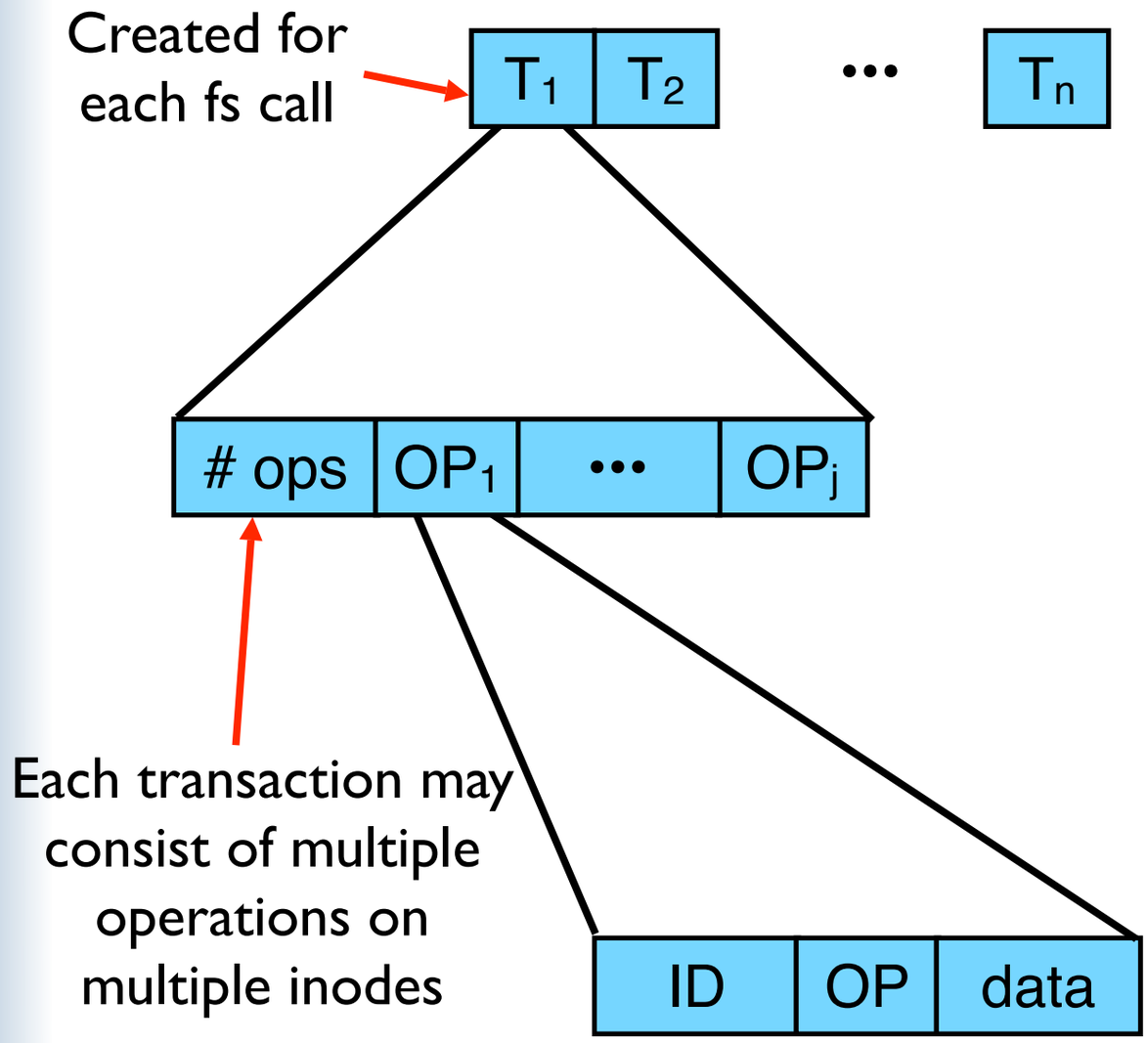
Data Page



Parity Page

Design — Transaction Log

Transaction Log



State Table

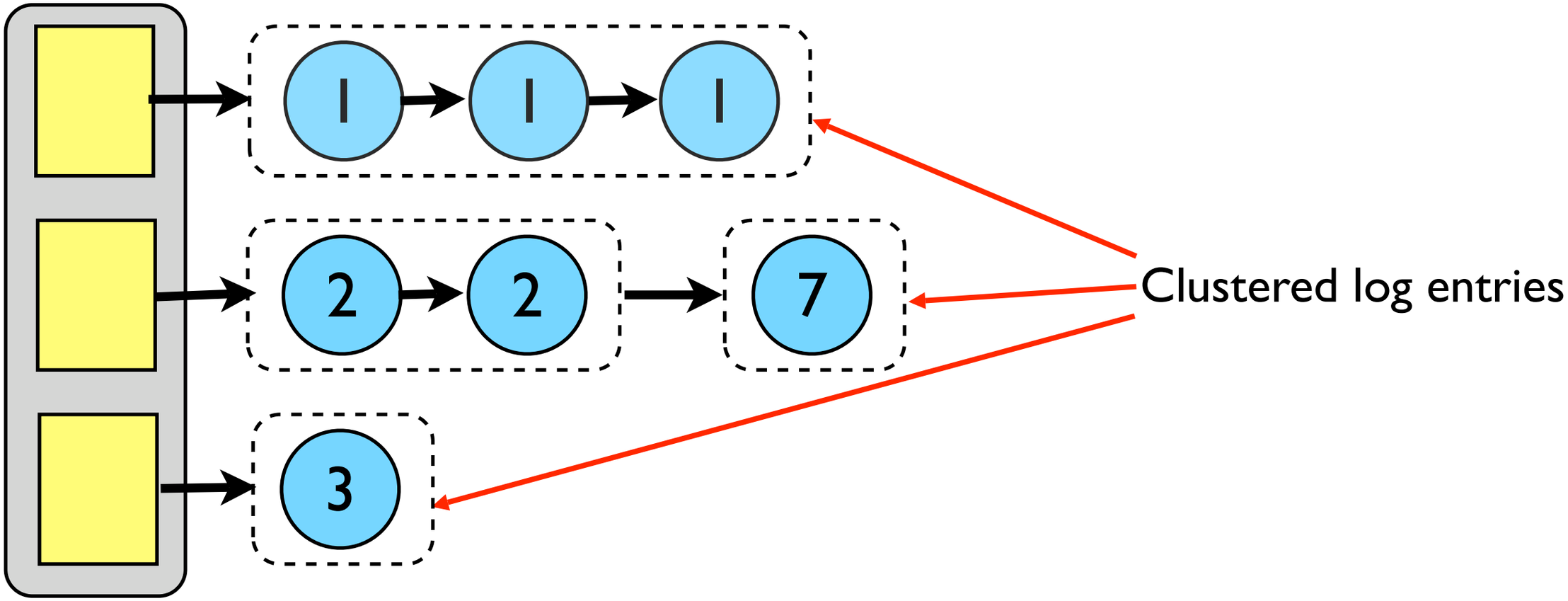
Index	State
<ID1>	S(ID1)
<ID2>	S(ID2)
•••	•••
<IDk>	S(IDk)

inode number

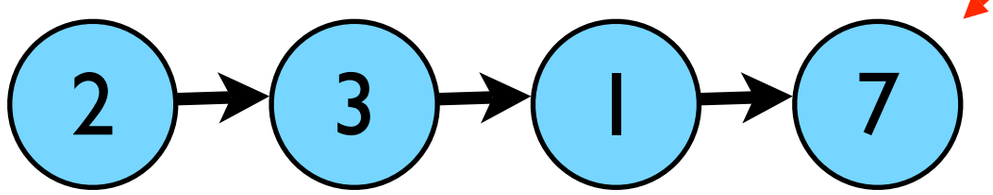
inode state

Consistency Checking

Transaction table (constructed from log)



Pending inode List



Consistency Checking Algorithm

- Initialize new log and state table
- Insert entries into transaction table and pending list
- For each inode on the pending list
  - If entry exists in "live" state table
    - Get next inode
  - If entry does not exist in the "live" state table
    - Fetch inode state from the "old" state table
    - Replay all ops in the transaction table
    - Compare replayed inode to "live" inode
  - Inconsistency if replay does not match "live" inode
- Requires O(1) lookup and replaying operations
- Block checks on NVRAM

ISSDM

Prototype Implementation

- Implemented in LiFS
- Implemented protected regions in MRAM allocator
  - Required additional calls in the interface
  - mprotect
  - EVENODD
- Logged metadata operations in LiFS calls
  - All except extents and xattrs
- Implemented On-Line Consistency Checker
  - Currently outside of LiFS, but can read LiFS log
  - Currently builds structures outside of protected area
  - Persistent structures can be stored in protected area

Prototype Performance

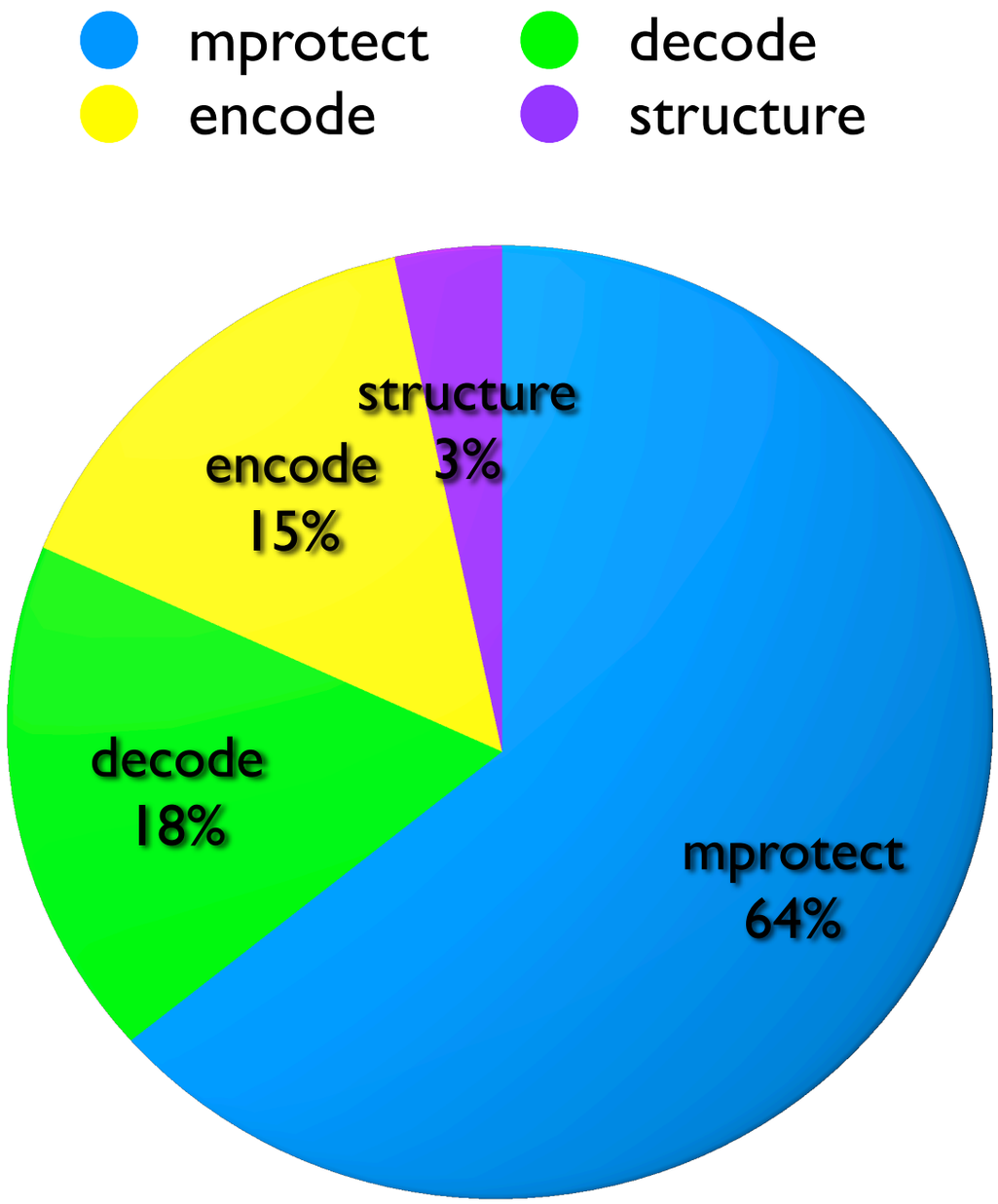
- Setup
  - 200 MB protected "MRAM" region
  - LiFS running through FUSE
  - EVENODD(96,64) and EVENODD(288,256)
- Fault Tolerance
  - Inject faults into protected region
  - Aggressively spawn threads that illegally write to region
  - Inject faults while performing valid writes to the region
- Raw write performance
  - Gives overhead for each prototype component
- File system write performance
  - Use a metadata write-centric workload
- Consistency Check Latency
  - Use same write-centric workload

Performance: Fault Tolerance

- Targeted entire 200 MB region
  - Performed 250,000 16-byte writes
  - Injected invalid writes while running workload
- Roughly 10,000 invalid writes were injected
  - 180-237 injected writes escaped page protection
  - 89-94% of these are caught by the EVENODD
  - Why not 100%?
    - Injected writes over-written by valid writes

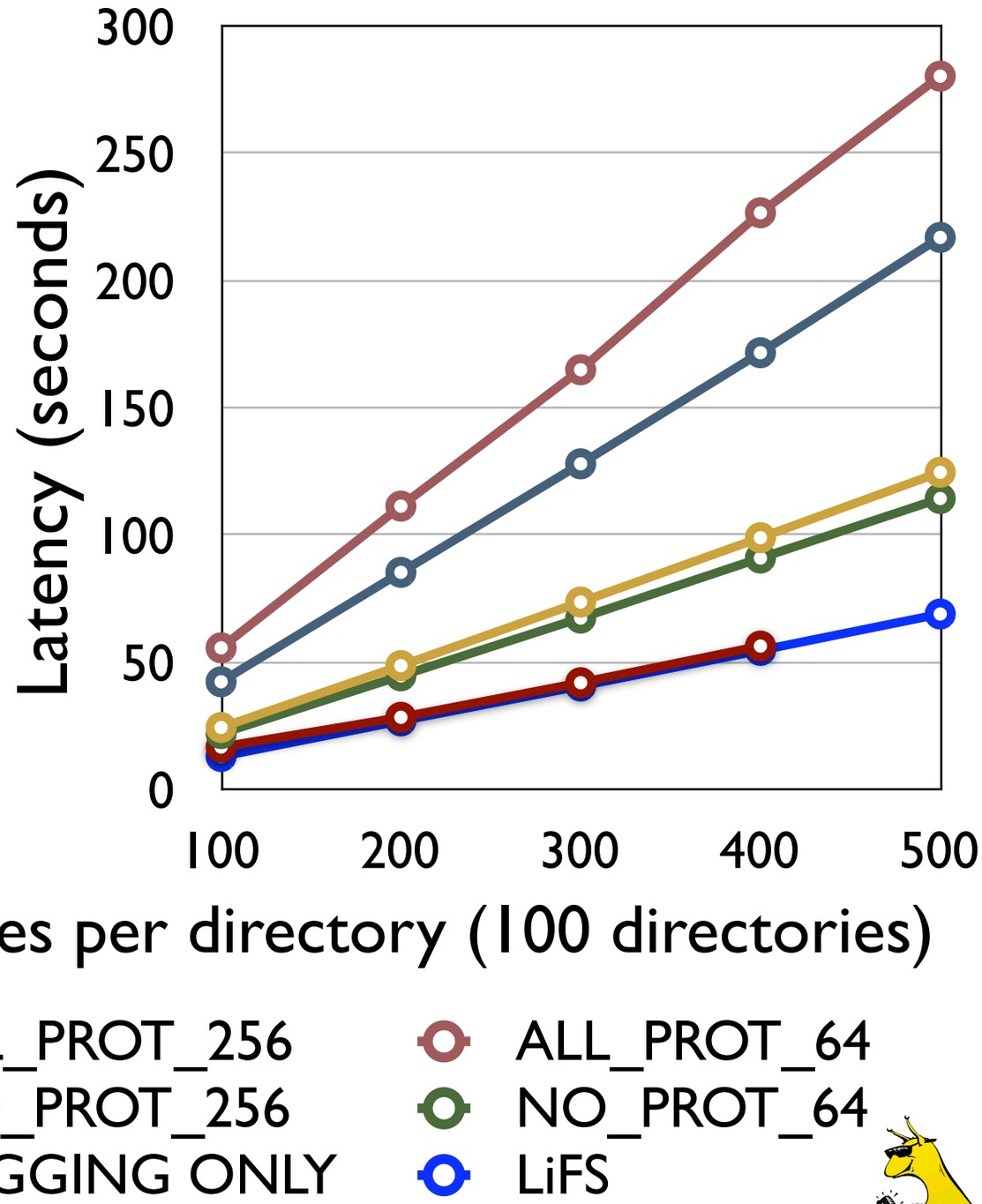
Performance: Raw Writes

- Most of the time is spent in mprotect
  - Mprotect: system call to protect memory
- Takes roughly 3 seconds to perform 250,000 writes
  - "Block" writes: small writes coalesced
- Without mprotect the same workload takes 1 second



File System Performance with Consistency Checking

- Create 100 directories
- Write 100-500 files to each directory
- Create links for each file
- Perm changes to file metadata
- Tests run with
  - Full protection
  - No mprotect
  - Only logging
  - Vanilla LiFS



Performance: On-Line Check

- Generated a log in LiFS
  - Same as latency workload (100 dirs/100-500 files)
  - Created 120,500–2.1 million operations
- Ran consistency checker outside of LiFS
  - Latency from less than 1 to 5.5 seconds

Related Work

- Queryable file systems
  - Expressive queries
  - Links don't have attributes (only files)
  - Advanced commercial file systems
    - Spotlight (Apple)
    - ZFS (Sun)
    - WinFS (eventually…)
  - Somewhat slow: information on disk
- In-memory file systems
  - Lack advanced file system features (links, searching)
  - No compression
  - Often treat memory as a block device (flash RAM)
  - Database research on utilizing persistent memory
- The Semantic Web

Future Work

- Push LiFS with these mechanisms into the kernel
- Distributed metadata using LiFS-based models
- RAID algorithms on multiple banks of MRAM
- Explore different approaches to data structures and compression
- Coding
  - Incorporate logging code for extents and x-attrs
  - Integrate the consistency checker into LiFS
  - Experiment with other ECCs
  - Allow communication between the MRAM-level mechanisms and the consistency checker

Conclusions

- MRAM promises big improvements for file systems
  - Richer metadata
  - Higher performance
  - Lower space requirements
  - Higher reliability
- Linking file system provides functionality currently unavailable
- Compression ensures that the metadata fits in MRAM
- Error correction and online consistency checking keep the metadata safe

ISSDM

Questions?

- Thanks to our sponsors:
  - National Science Foundation
  - SSRC industrial sponsors
  - LANL

- Thanks to MRAM team members
  - Carlos Maltzahn, Scott Brandt
  - Sasha Ames, Nikhil Bobb, Karl Brandt, Nate Edel, Kevin Greenan, Adam Hiatt, Owen Hofmann, Alisa Neeman, Mark Storer, Deepa Tuteja

- http://www.ssrc.ucsc.edu/proj/scm.html